Test Driven Development with Boost.Test

Wolfram Rösler

Slides: `https://gitlab.com/wolframroesler/Talks/blob/master/tdd.pdf`
Example code: `https://gitlab.com/wolframroesler/boost-test-demo`

The Software Development Workflow

a. k. a. the waterfall model

1. Have a specification
2. Implement it
3. Test it
4. Ship it
5. Be happy
6. Go to next project

The Software Development Workflow

what's it's like in reality

1. Have a rough idea what to do
2. Implement it because that's fun
3. No need to test because I'm a genius
4. Ship it
5. Fix bugs and regressions
6. Go to 5

The Test Driven Development Workflow

in a nutshell:

TEST FIRST

The Test Driven Development Workflow

1. Write test program
2. Implement until it passes
3. Refactor
4. Make all tests pass

Benefits of TDD

Test coverage

Confidence

Less debugging

Few regressions

Release anytime

Pays out from the start

Downsides of TDD

Takes time

No quick changes

Requires dedication and effort

Who tests the test program?

False negatives

Not always easy

What You Need For TDD

1. Tools

Easy: cppunit, Boost.Test, Google Test, whatever

2. Processes

Less easy: Integrate TDD into dev workflows

3. Mindset

Difficult: Change habits

Test Granularity

Unit Test

← TDD happens here

the smallest testable thing

Module Test

module = many units

Integration Test

interaction between modules

System Test

everything

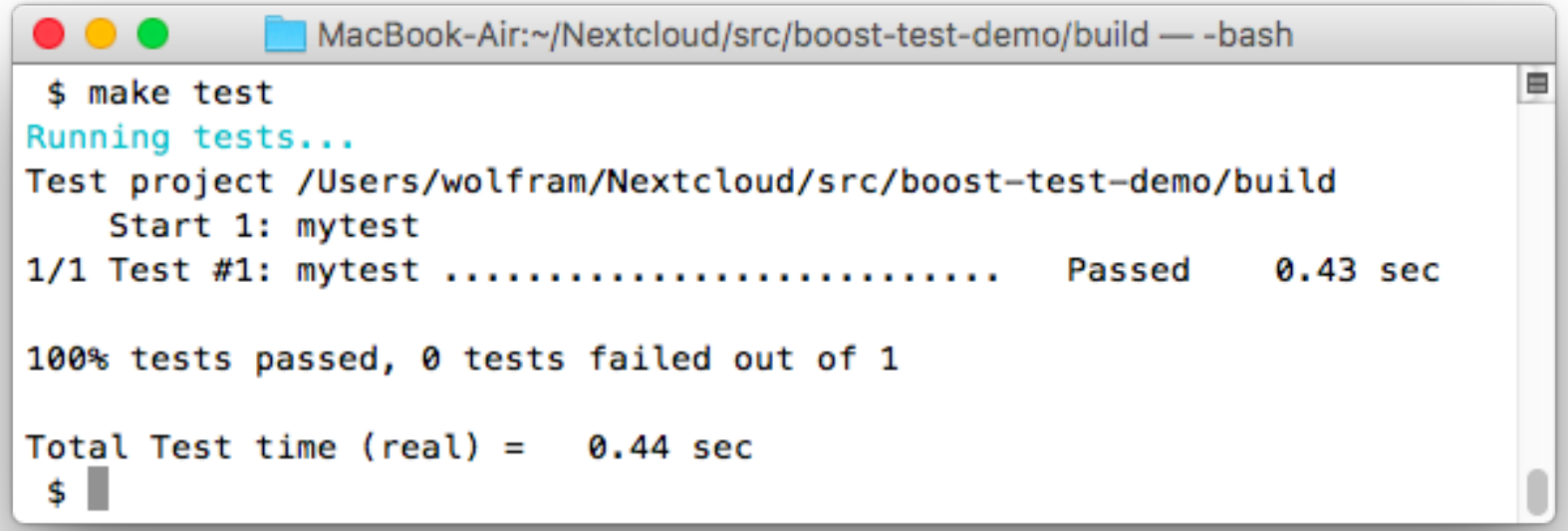
Test Organization

Test assertion	A single comparison
Test case	Several assertions
Test suite	Several test cases
Test plan	Several test suites

Anatomy of a Test Case

1. Set up
2. Do something
3. Validate
4. Tear down

Live Demo: Boost.Test



https://gitlab.com/wolframroesler/boost-test-demo

Ways To Test Something

White-box Test	You know it
Black-box Test	You don’t know it
Boundary Test	Edge cases
Regression Test	Bugs
Fuzz Test	Random input

Not Easily Testable

People

Databases

Web servers

Hardware

Concurrency

How To Test It Anyhow

Simulators

Mock-ups

Error simulation

Plenty of imagination

Cheat Sheet: Classical TDD Rules

- Test first, then implement, finally refactor.
- Implement until tests pass, but no further.
- Write the simplest possible code to make a test pass, then refactor to improve code quality.
- Implement until new tests pass, then refactor until old tests pass.
- Unless refactoring, don’t change anything unless it’s required to make a failing test pass.

Cheat Sheet: Getting Started

- Assume that the code you have is correct.
- Establish TDD tools and processes.
- The next time a feature is added, test first.
- The next time a bug is found, reproduce it in a test case.
- Add test cases for existing functionality as required.
- Don’t modify any code unless the modification is covered by a test case.

Resources

https://**en.wikipedia.org**/wiki/Test-driven_development

https://**en.wikipedia.org**/wiki/Unit_testing

https://**en.wikipedia.org**/wiki/XUnit

http://**wiki.c2.com**/?TestDrivenDevelopment

https://**www.boost.org**/doc/libs/1_68_0/libs/test/doc/html/index.html

https://**stackoverflow.com**/questions/64333/disadvantages-of-test-driven-development

https://**gitlab.com**/wolframroesler/afl-demo – Fuzz test demo program